Compiling ML models

Max Bernstein

Hi, I’m Max

- PhD student working with Frank Tip on compilers
- Before that, worked at Facebook for 5 years on compilers for Python
- Bikes
- Rock climbing
- Baking
- etc

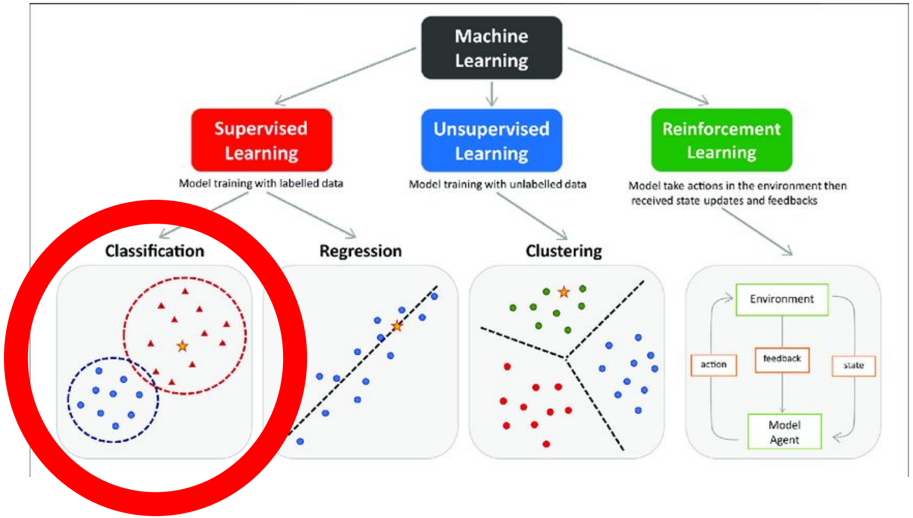
Caveat

I don’t know anything about ML. If you see something you find misleading or incorrect, please say something!

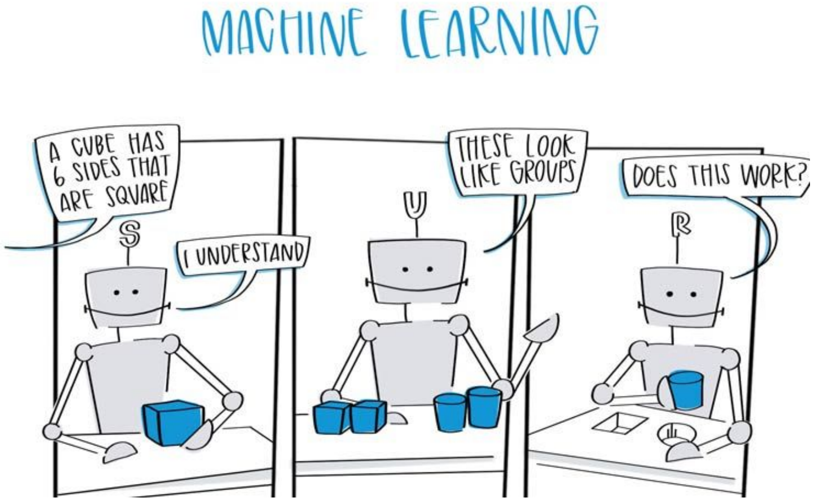
Also in general, **ask questions as you have them**.

What is machine learning?

- Continuously adjusting a function (model) to get the results you want
- Generally, getting the computer to adjust the function for you



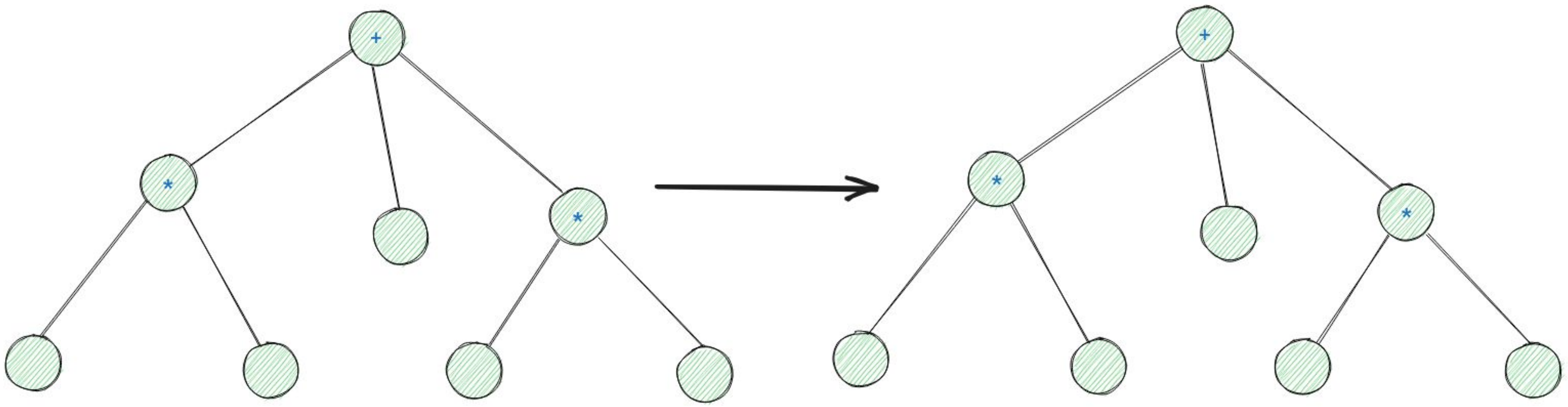
https://www.researchgate.net/figure/The-main-types-of-machine-learning-Main-approaches-include-classification-and_fig1_354960266



https://ceralytics.com/3-types-of-machine-learning/

What is a compiler?

- A function that takes in a program and outputs a program
- Can be the same or different representation



What is a compiler?

- A function that takes in a program and outputs a program

Source

```
/tmp/tmpjc34798v/explorer_lib.py
3 def test(a: int32):
4   if a:
5     return 1
```

Text!

Bytecode

```
bb0                                                                  -
    (4) 0: LOAD_LOCAL 1: (0, ('__static__', 'int32', '#'))
    (4) 2: POP_JUMP_IF_ZERO 4
    (5) 4: LOAD_CONST 2: 1
    (5) 6: RETURN_VALUE 0
    (6) 8: LOAD_CONST 3: 2
    (6) 10: RETURN_VALUE 0
```

Bytecode!

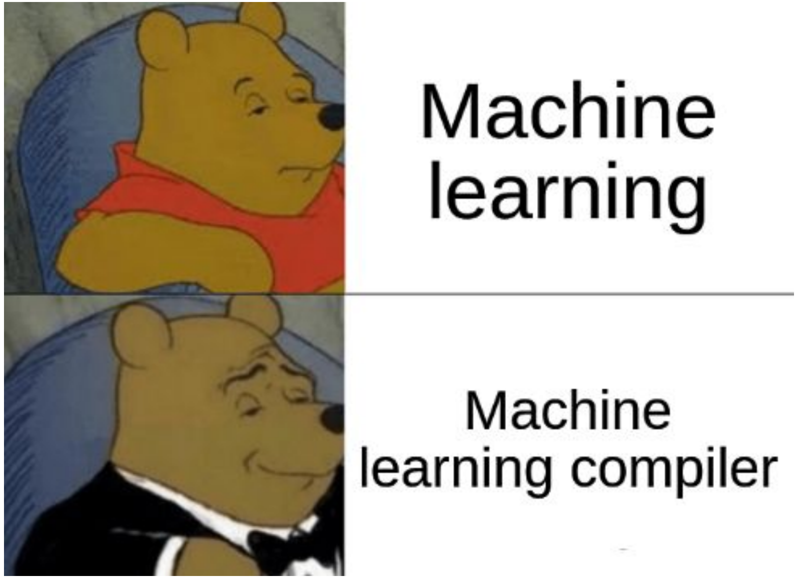
Assembly

```
StaticLinkFrame                                                      +
StaticEntryPoint                                                     +
Reentry                                                              +
Generic en                                                           +
Link Frame                                                           +
Native entry                                                         +
(4) CondBranch<1, 2> v3
      (4) 0x7f10231b41b6: test esi,esi                               -
      (4) 0x7f10231b41b8: jne 0x7f10231b41cd
(6) v5:ImmortalLongExact[2] = LoadConst<ImmortalLongExact[2]>
      (6) 0x7f10231b41be: movabs rax,0x7f10233b4110                  -
      (6) 0x7f10231b41c8: jmp 0x7f10231b41d7
(5) v4:ImmortalLongExact[1] = LoadConst<ImmortalLongExact[1]>
      (5) 0x7f10231b41cd: movabs rax,0x7f10233b40f0                  -
Epilogue                                                             +
Epilogue (restore regs; pop native frame; error exit)                +
--unassigned--                                                       +
```

Assembly!

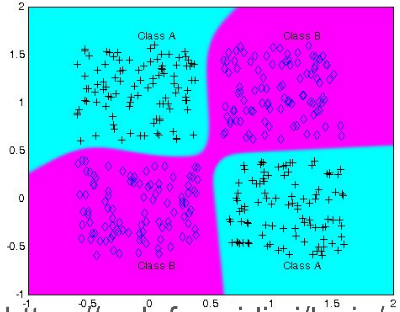
A new (to me) frontier for compilers: ML

- It’s all two of my friends talk about
- I should be able to communicate with them
- Might as well build something
- Let’s build a compiler for micrograd



micrograd: a scalar-valued neural network library

- By Andrej Karpathy
- Here is a training loop to learn XOR function on [0,1]
- Iteratively improve model
- Just 200LOC



https://web.fs.uni-lj.si/lasin/wp-content/include-me/neural/nn04_mlp_xor/

```
from micrograd.nn import MLP
model = MLP(2, [4, 1])
batch = [([0, 0], 0), ([0, 1], 1), ([1, 0], 1), ([1, 1], 0)]
batch_size = len(batch)
for epoch in range(1000):
    model.zero_grad()
    epoch_loss = 0.
    for xs, expected in batch:
        output = model(xs)
        loss = (output-expected)**2
        epoch_loss += loss.data
        loss.backward()
    for p in model.parameters():
        p.data -= 0.1 * p.grad / batch_size
    epoch_loss /= batch_size
    print(f"...epoch {epoch:4d} loss {epoch_loss:.4f}")

for im in batch:
    print(model(im[0]))
```

Forward

Backward

Update

micrograd: a scalar-valued neural network library

- Test results: looking good
- Nearly instant: 0.2 seconds
- Small network: only 53 nodes

0 XOR 0

0 XOR 1

1 XOR 0

1 XOR 1

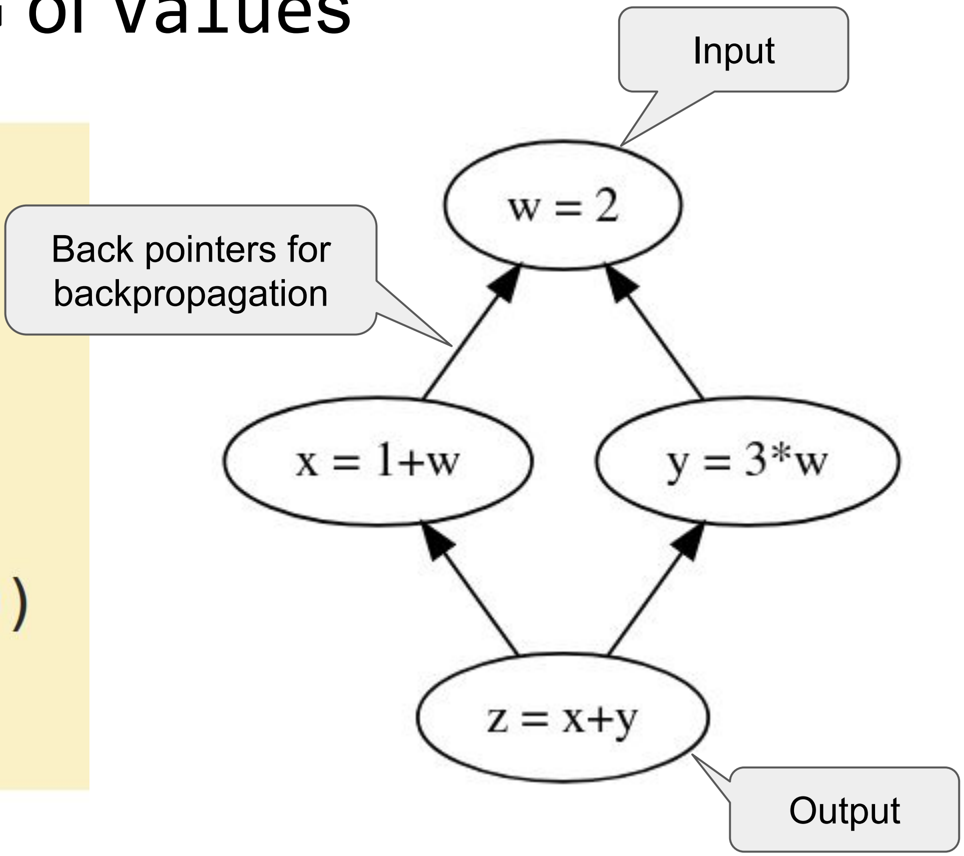
```
...epoch  996 loss 0.0000
...epoch  997 loss 0.0000
...epoch  998 loss 0.0000
...epoch  999 loss 0.0000
Value(data=7.390300937258563e-12, grad=0)
Value(data=0.9999999999973999, grad=0)
Value(data=0.9999999999975436, grad=0)
Value(data=1.3365540833096393e-12, grad=0)
```

micrograd does math

```
>>> w = Value(2)
>>> x = 1 + w
>>> y = 3 * w
>>> z = x + y
>>> z
Value(data=9, grad=0)
>>>
```

micrograd builds a DAG of `Values`

```
>>> w = Value(2)
>>> x = 1 + w
>>> y = 3 * w
>>> z = x + y
>>> z
Value(data=9, grad=0)
>>>
```



We need the back pointers for backpropagation

```
class Value:
    # ...
    def __mul__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other), '*')

        def _backward():
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out._backward = _backward

        return out
```

back pointers in _prev

A closure to adjust gradients using the chain rule

And we iterate over (reversed) topological sort

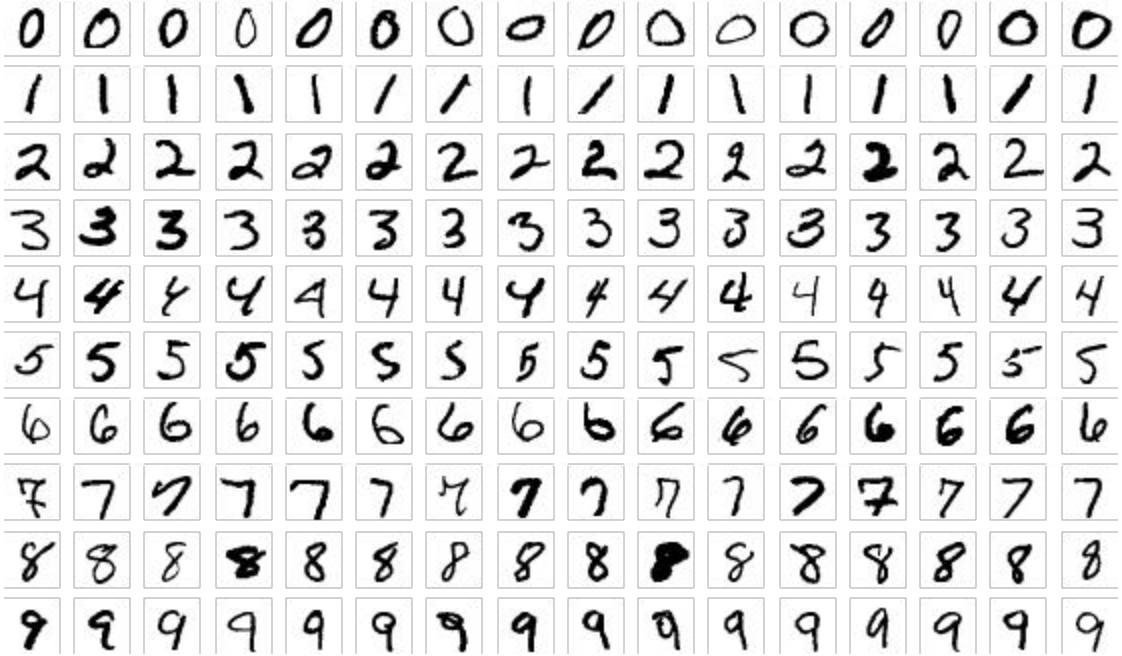
```
class Value:
    # ...
    def backward(self):

        # topological order all of the children in the graph
        topo = []
        visited = set()
        def build_topo(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topo(child)
                topo.append(v)
        build_topo(self)

        # --- the new bit ---
        # go one variable at a time and apply the chain rule to get its gradient
        self.grad = 1
        for v in reversed(topo):
            v._backward()
```

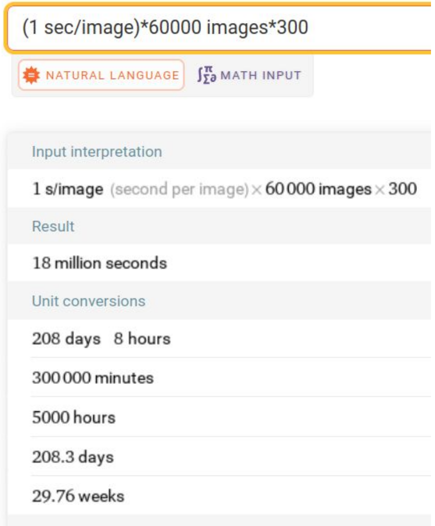
Does it scale?

- Let's learn something else. How about MNIST handwriting dataset?
- Input size is 784 (28x28) instead of 2
- And each input is [0,255] instead of [0,1]

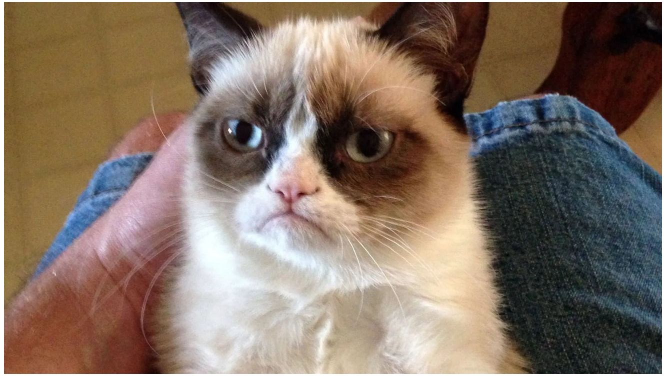


It does not scale

- Forward+backward pass for *one image* is 1 second
- Train on the entire 60k image library 300 times?



Hmmm.

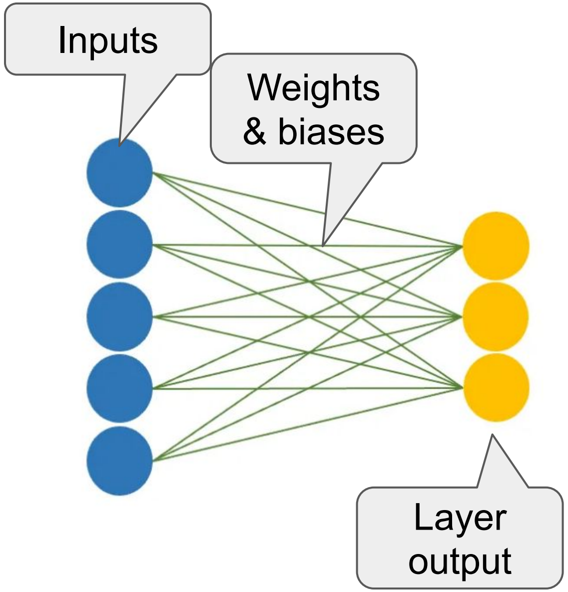


RIP Grumpy Cat

What's going on? Two views of neural networks

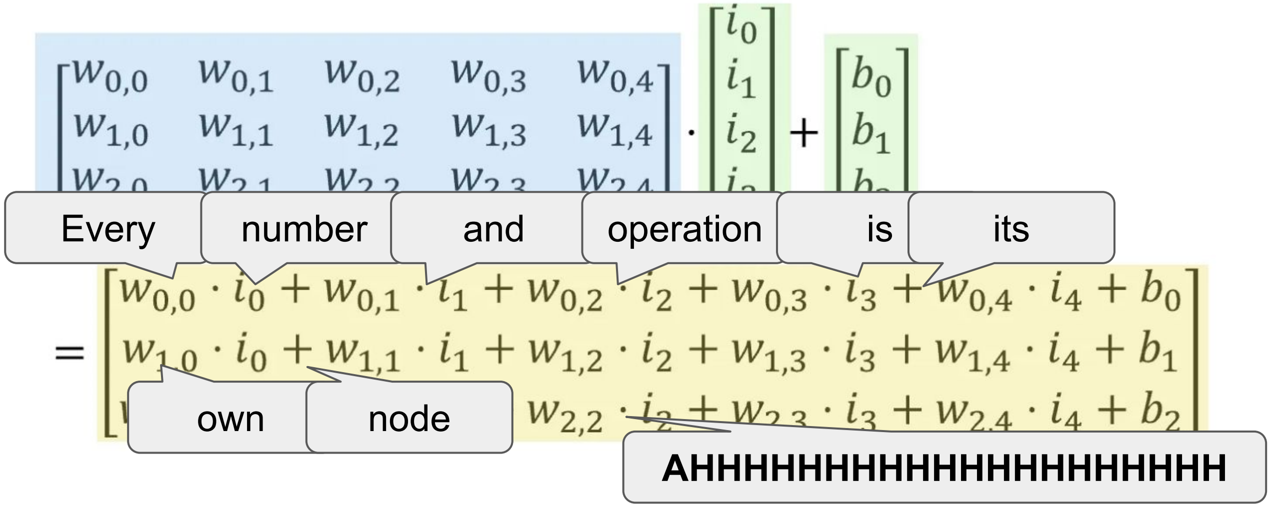
$$\underbrace{\begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & w_{0,3} & w_{0,4} \\ w_{1,0} & w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,0} & w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \end{bmatrix}}_{\text{Weights}} \cdot \underbrace{\begin{bmatrix} i_0 \\ i_1 \\ i_2 \\ i_3 \\ i_4 \end{bmatrix}}_{\text{Inputs}} + \underbrace{\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}}_{\text{Biases}}$$

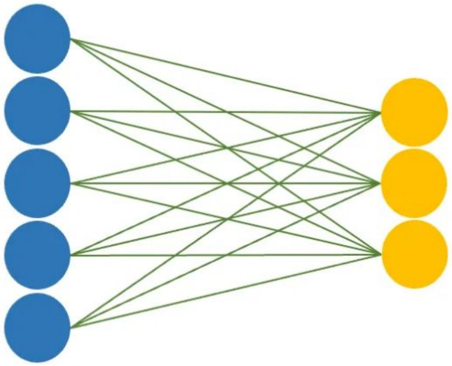
$$= \underbrace{\begin{bmatrix} w_{0,0} \cdot i_0 + w_{0,1} \cdot i_1 + w_{0,2} \cdot i_2 + w_{0,3} \cdot i_3 + w_{0,4} \cdot i_4 + b_0 \\ w_{1,0} \cdot i_0 + w_{1,1} \cdot i_1 + w_{1,2} \cdot i_2 + w_{1,3} \cdot i_3 + w_{1,4} \cdot i_4 + b_1 \\ w_{2,0} \cdot i_0 + w_{2,1} \cdot i_1 + w_{2,2} \cdot i_2 + w_{2,3} \cdot i_3 + w_{2,4} \cdot i_4 + b_2 \end{bmatrix}}_{\text{Layer output}}$$



Andre Ye
https://towardsdatascience.com/if-rectified-linear-units-are-linear-how-do-they-add-nonlinearity-40247d3e4792

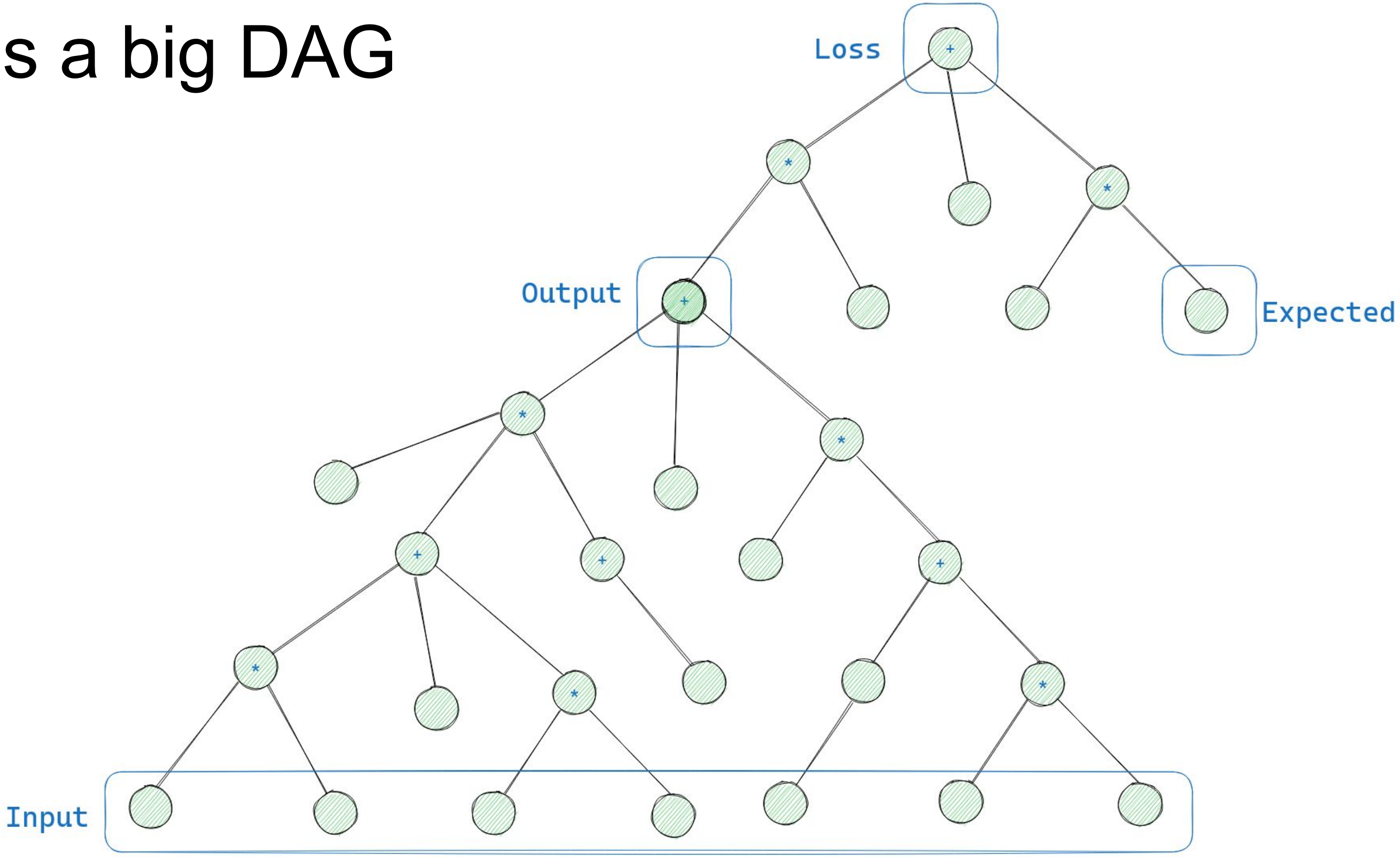
micrograd does this manually





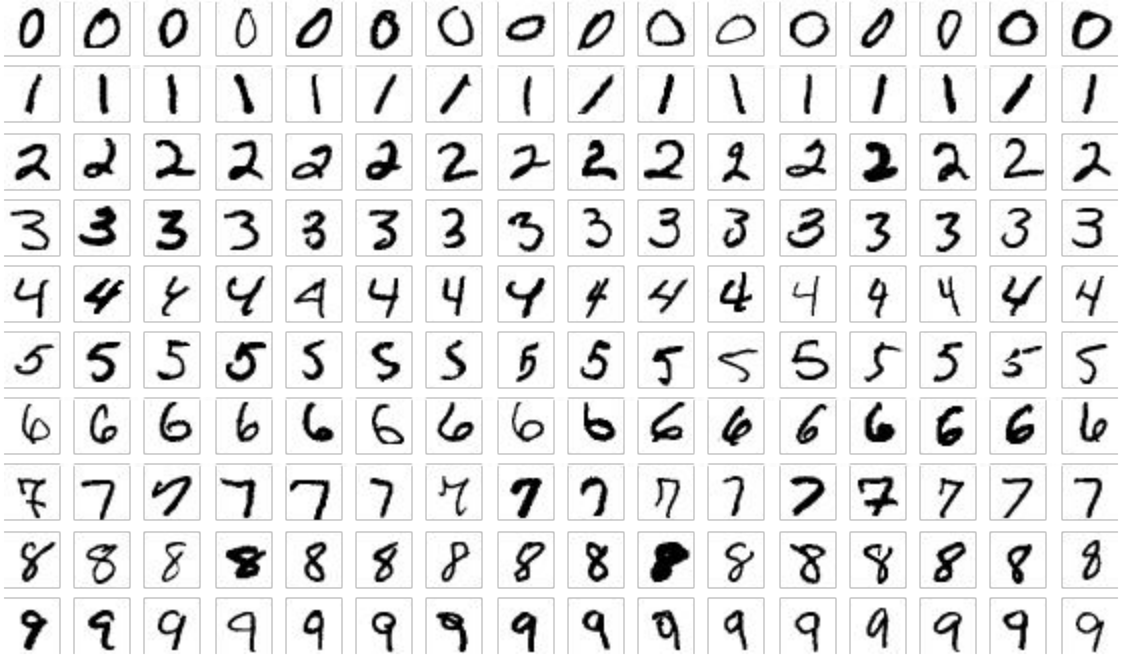
Andre Ye
https://towardsdatascience.com/if-rectified-linear-units-are-linear-how-do-they-add-nonlinearity-40247d3e4792

That's a big DAG



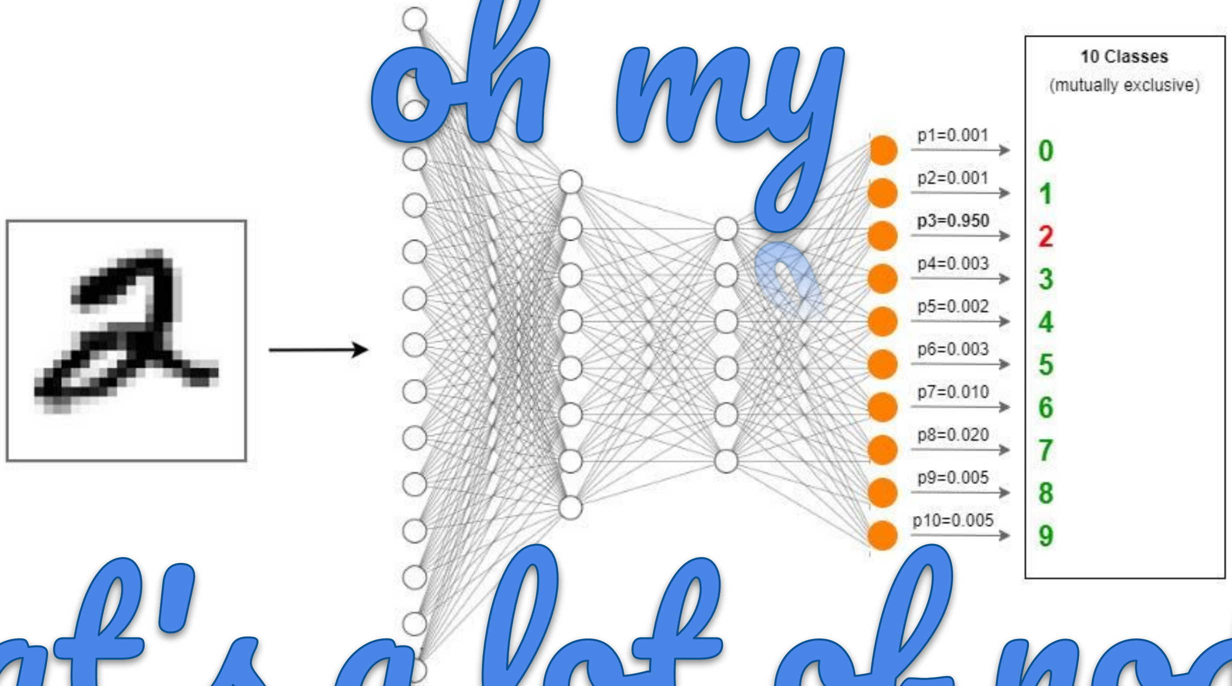
What that graph size means for us

- MNIST network is `MLP(784, [50, 10])`
- Which is 120k nodes
- ...allocated every time
- ...traversed every time
- ...even though the graph never changes
- ...in Python



By Suvanjanprasai - Own work,
https://commons.wikimedia.org/w/index.php?curid=132282871

oh my

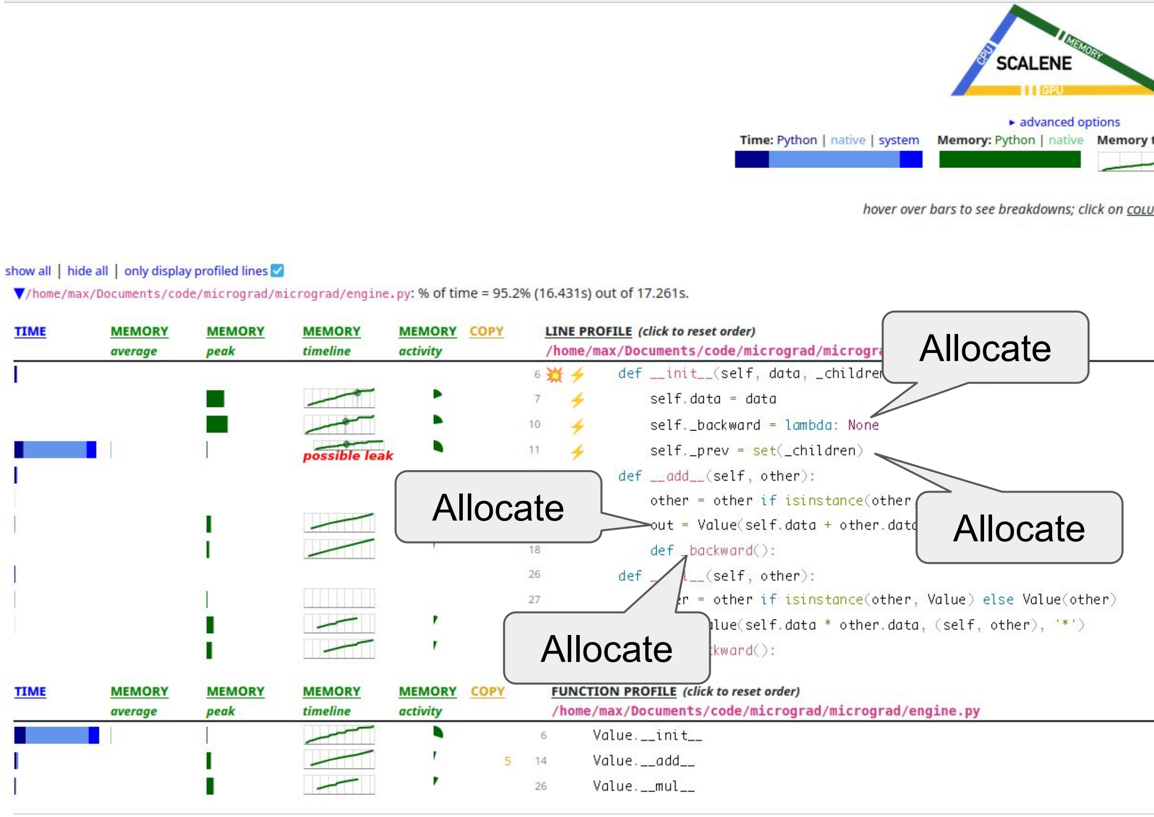


that's a lot of nodes

Rukshan Pramoditha
https://towardsdatascience.com/creating-a-multilayer-perceptron-mlp-classifier-model-to-identify-handwritten-digits-9bac1b16fe10

What is slow?

- Profile with Scalene
- Scalene says `Value` is a hotspot
- That's a lot of allocation



- Also, topological sort

Backward too!

What do we know doesn’t change?

- Graph structure/topology
- Traversal order

Solution: Linearize the forward and backward traversals **ahead of time**

```python
from micrograd.engine import Value
from micrograd.nn import MLP
model = MLP(784, [50, 10])
in_ = [Value(0) for _ in range(784)]
out_ = model(in_)
joined = loss(out_)
topo = joined.topo()

# set input
for idx, pixel in enumerate(image):
    in_[idx].data = pixel
# forward
for node in topo:
    node._forward()  # re-calculate .data
# backward
loss.grad = 1
for node in reversed(topo):
    node._backward()
```

This is for one image input

We can still do better

- There is a lot of Python overhead
- We could try using a JIT or AOT Python compiler...
  - (Have you read my blog post? Compiling dynamic languages is *hard*)
- ...or, we could write our own compiler

A small compiler

```
class Value:
    # ...
    def var(self):
        return f"data[{self._id}]"

    def set(self, val):
        return f"{self.var()} = {val};"

    def compile(self):
        if self._op in ('weight', 'bias', 'input'):
            # Not calculated; set elsewhere
            return ""
        if self._op == '':
            return self.set(f"{self.data}")
        if self._op == '*':
            c0, c1 = self._prev
            return self.set(f"{c0.var()}*{c1.var()}")
```

Not *exactly* normal tree-walking

```
>>> from micrograd.engine import Value
>>> x = Value(1)
>>> y = Value(2)
>>> z = x * y
>>> order = z.topo()
>>> for v in order:
...     print(v.compile())
...
data[1] = 2;
data[0] = 1;
data[2] = data[1]*data[0];
>>>
```

Compile the models to C

```python
from micrograd.engine import Value
from micrograd.nn import MLP
model = MLP(784, [50, 10])
in_ = [Value(0) for _ in range(784)]
out_ = model(in_)
joined = loss(out_)
topo = joined.topo()

# set input
print("""void set_input(double* pixels) {
for (int i = 0; i < 784; i++) {
    data[input_start+i] = pixels[i];
}
}""")
# forward
print("void forward() {")
for node in topo:
    print(node.compile_forward())
print("}")
# backward
print("void backward() {")
print("grads[loss_idx] = 1.0;")
for node in reversed(topo):
    print(node.compile_backward())
print("}")
```

```python
1 from micrograd.engine import Value
2 from micrograd.nn import MLP
3 model = MLP(784, [50, 10])
4 in_ = [Value(0) for _ in range(784)]
5 out_ = model(in_)
6 joined = loss(out_)
7 topo = joined.topo()
8
9 # set input
10 for idx, pixel in enumerate(image):
11     in_[idx].data = pixel
12 # forward
13 for node in topo:
14     node._forward()  # re-calculate .data
15 # backward
16 loss.grad = 1
17 for node in reversed(topo):
18     node._backward()
```

1-7 same!

```python
1 from micrograd.engine import Value
2 from micrograd.nn import MLP
3 model = MLP(784, [50, 10])
4 in_ = [Value(0) for _ in range(784)]
5 out_ = model(in_)
6 joined = loss(out_)
7 topo = joined.topo()
8
9 # set input
10 print("""void set_input(double* pixels) {
11 for (int i = 0; i < 784; i++) {
12     data[input_start+i] = pixels[i];
13 }
14 }""")
15 # forward
16 print("void forward() {")
17 for node in topo:
18     print(node.compile_forward())
19 print("}")
20 # backward
21 print("void backward() {")
22 print("grads[loss_idx] = 1.0;")
23 for node in reversed(topo):
24     print(node.compile_backward())
25 print("}")
```

A side-by-side look

A side-by-side look

```python
from micrograd.engine import Value
from micrograd.nn import MLP
model = MLP(784, [50, 10])
in_ = [Value(0) for _ in range(784)]
out_ = model(in_)
joined = loss(out_)
topo = joined.topo()

# set input
for idx, pixel in enumerate(image):
    in_[idx].data = pixel
# forward
for node in topo:
    node._forward()  # re-calculate .data
# backward
loss.grad = 1
for node in reversed(topo):
    node._backward()
```

```python
from micrograd.engine import Value
from micrograd.nn import MLP
model = MLP(784, [50, 10])
in_ = [Value(0) for _ in range(784)]
out_ = model(in_)
joined = loss(out_)
topo = joined.topo()

# set input
print("""void set_input(double* pixels) {
for (int i = 0; i < 784; i++) {
    data[input_start+i] = pixels[i];
}
}""")
# forward
print("void forward() {")
for node in topo:
    print(node.compile_forwa
print("}")
# backward
print("void backward() {")
print("grads[loss_idx] = 1.0;")
for node in reversed(topo):
    print(node.compile_backward())
print("}")
```

Data just used for graph scaffolding

Looping happens **in the compiler**

Example output

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <Python.h>
    double data[40857];
    double grad[40857];
    static inline __attribute__((always_inline)) double relu(double x) {
        return fmax(x, 0);
    }
    void init() {

data[0] = 0.23550571390294128L;
data[1] = 0.06653114721000164L;
data[2] = -0.26830328150124894L;
data[3] = 0.1715747078045431L;
data[4] = -0.6686254326224383L;
data[5] = 0.6487474938152629L;
data[6] = -0.23259038277158273L;
data[7] = 0.5792256498313748L;
data[8] = 0.8434530197925192L;
data[9] = -0.3847332240409951L;
data[10] = 0.9844941451716409L;
data[11] = -0.5901079958448365L;
data[12] = 0.31255526637777775L;
```

...

...

```
data[39753] = 0.25806281413875665L;
data[39754] = -0.6964206879256336L;
data[39755] = 0.06098572366796007L;
data[39756] = 0.3613208562489416L;
data[39757] = 0.07010011879704026L;
data[39758] = -0.32217329468233746L;
data[39759] = 0L;
}
    void set_input(PyObject* input_data) {
        const char* buf = PyBytes_AsString(input_data);
        if (buf == NULL) {
            abort();
        }
        for (int i = 0; i < 784; i++) {
            data[39760+i] = ((double)(unsigned char)buf[i])/255;
        }
    }

void forward() {
```

Example outpu

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <Python.h>
    double data[40857];
    double grad[40857];
    static inline __attribute__((always_
        return fmax(x, 0);
    }
    void init() {

data[0] = 0.23550571390294128L;
data[1] = 0.06653114721000164L;
data[2] = -0.26830328150124894L;
data[3] = 0.1715747078045431L;
data[4] = -0.6686254326224383L;
data[5] = 0.6487474938152629L;
data[6] = -0.23259038277158273L;
data[7] = 0.5792256498313748L;
data[8] = 0.8434530197925192L;
data[9] = -0.3847332240409951L;
data[10] = 0.9844941451716409L;
data[11] = -0.5901079958448365L;
data[12] = 0.31255526637777775L;
```

...

```
void forward() {
data[40544] = data[0]*data[39760];
data[40546] = data[40544]+0;
data[40547] = data[1]*data[39761];
data[40548] = data[40546]+data[40547];
data[40549] = data[2]*data[39762];
data[40550] = data[40548]+data[40549];
data[40551] = data[3]*data[39763];
data[40552] = data[40550]+data[40551];
data[40553] = data[4]*data[39764];
data[40554] = data[40552]+data[40553];
data[40555] = data[5]*data[39765];
data[40556] = data[40554]+data[40555];
data[40557] = data[6]*data[39766];
data[40558] = data[40556]+data[40557];
data[40559] = data[7]*data[39767];
data[40560] = data[40558]+data[40559];
data[40561] = data[8]*data[39768];
data[40562] = data[40560]+data[40561];
data[40563] = data[9]*data[39769];
data[40564] = data[40562]+data[40563];
data[40565] = data[10]*data[39770];
data[40566] = data[40564]+data[40565];
data[40567] = data[11]*data[39771];
data[40568] = data[40566]+data[40567];
```

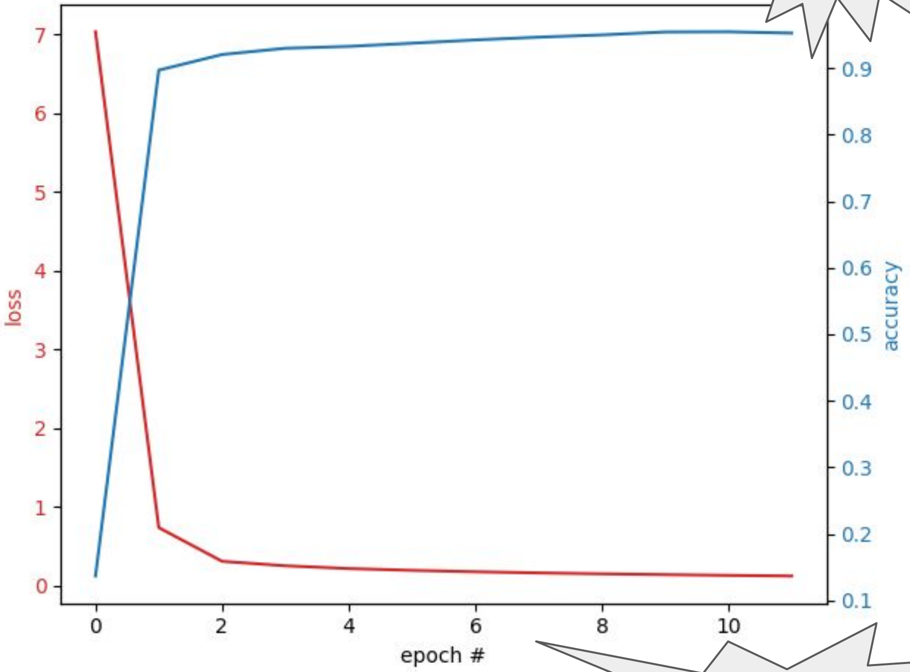
```
(input_data);
gned char)buf[i])/255;
```

ew.

Does it work?

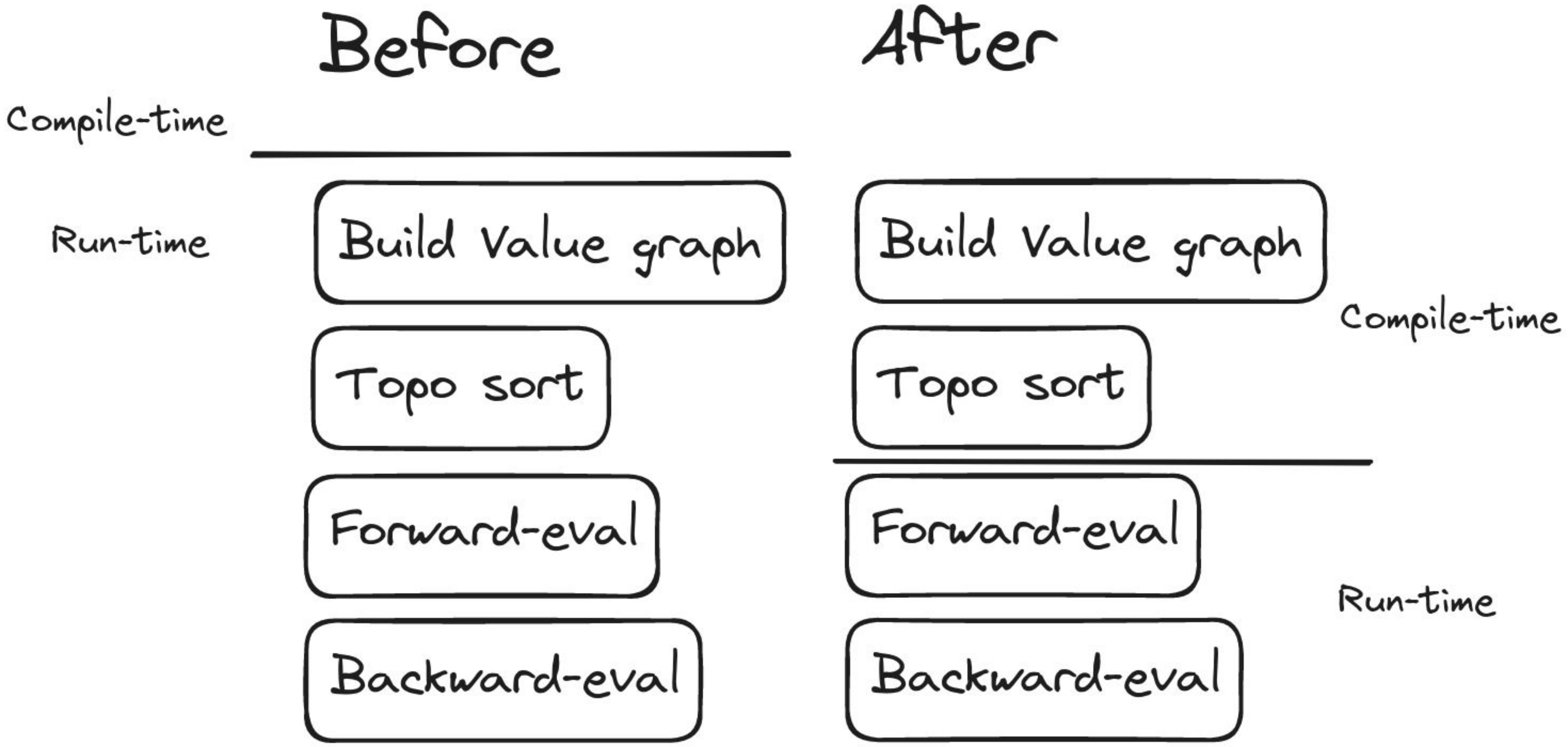
- Looks like we are training alright
  - Accuracy is checked on a different (“test”) dataset
- Should probably write unit tests

95%



Just over 1 minute!

For each image...



Gotta go fast

	Compile time (s)	Time per epoch (s)	Speedup
Interpreted	0	60,000	1x
TCC	0.5	45	1333x
Clang `-O0`	~30	30	2000x
Clang `-O1`	~350	8	7500x

Conclusion

micrograd: 200LOC

micrograd+compiler: 280LOC

speedup: absolutely hog wild

Didn't even need to change the neural network code! Very extensible!!

Call me, beep me!

Web: bernsteinbear.com

This slide deck